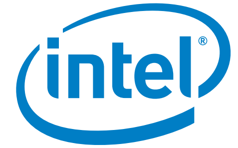
APPLICATION NOTE

Intel Corporation
Software Defined Datacenter Solutions Group



Multiple Network Interfaces in Kubernetes

Table of Contents

1.0 Introduction 1
2.0 Multiple networking interfaces using Multus 2
3.0 Improving network performance using Multus & SR-IOV/DPDK 3
4.0 Configuring Multus in Kubernetes 5
4.1 Configure Multus using network objects 5
4.2 Configure Multus using config file 9
4.3 Verifying pod networks 10
5.0 Configuring SR-IOV/DPDK in Kubernetes 11
5.1 Enable SR-IOV 11
6.0 Conclusion 13
7.0 Appendix 13
7.1 How to define TPR-based network objects 13
7.2 Hardware 13
7.3 Software 14
7.4 Terminology 14
7.5 Reference documents 15

1.0 Introduction

For some time, the Communications Service Provider (CommSP) industry has been moving to embrace the developments in Software Defined Networking (SDN) and Network Function Virtualization (NFV). Among the reasons for this are improved service deployment agility, operational efficiencies and reduced infrastructure costs. Initially, this move involved the virtualization of the physical network functions into virtual machines (VM). More recently, the next phase of this evolution has begun with the adoption of containers as a means to achieve greater scalability and resiliency. These new network functions are often referred to as Cloud Native Virtualized Network Functions (VNF).

Kubernetes is the leading container orchestration engine (COE). It is an open source system for automating deployment, scaling, and management of containerized applications. Kubernetes was developed at Google and is the anchor project in the Cloud Native Computing Foundation (CNCF), which is governed by the Linux Foundation (LF).

However, Kubernetes lacks the required functionality to provide and support multiple network interfaces in VNF's. Traditionally, multiple network interfaces are employed by network functions to provide for separation of control, management and data/user network planes. They are also used to support different protocols or software stacks and different tuning and configuration requirements.

This document introduces a solution from Intel for Kubernetes called Multus, which addresses this need. Multus is a container network interface (CNI) plugin that can be used to create multiple network interfaces for pods in Kubernetes. A pod is a deployable unit of computing and is created and managed by Kubernetes. Multus is designed and developed to enable easier migration of current NFV use cases to a container environment. Figure 1 shows Kubernetes networking before and after Multus.

In the longer term, Intel believes that a native Kubernetes approach for providing multiple network interfaces is required. As Intel and partners continue to work to support the journey to cloud native, many options are being evaluated within the Kubernetes community.

This document is designed for engineers working with Kubernetes who want a deep dive into how to configure Multus for VNFs. It will also cover

- how Multus can utilize single root I/O virtualization (SR-IOV) for accelerating the data/user plane throughput for high packet throughput and low processing latency.
- the open source software components required to utilize the Multus CNI plugin.

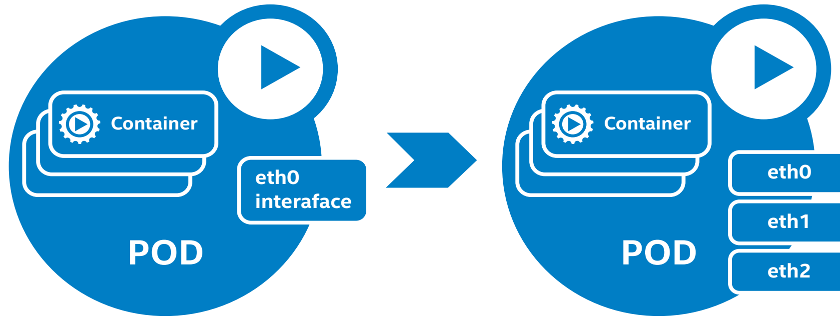


Figure 1. Kubernetes networking before and after Multus

Note: The document does not describe how to setup a Kubernetes cluster. It is assumed that this is available to the reader before undertaking the steps in this document. For more setup and installation guidelines for a complete system please refer to the document in table 7.5 of the Appendix called "Deploying Kubernetes and Container Bare Metal Platform for NFV Use Cases with Intel® Xeon® Scalable Processors."

This document is part of the Container Experience Kit for Kubernetes Networking. The container experience kits are collections of user guides, application notes, feature briefs and other collateral that provide a library of best-practice documents for engineers who are developing container-based applications. Other documents in this Experience Kit include:

Document Title	Document Type	Location URL
Multiple Network Interface Support in Kubernetes	Feature Brief	https://builders.intel.com/docs/networkbuilders/multiple-network-interfaces-support-in-kubernetes-feature-brief.pdf

2.0 Multiple networking interfaces using Multus

Multus is a CNI plugin specifically designed to provide support for multiple networking interfaces in a Kubernetes environment. CNI, a CNCF project, is a specification and supporting framework for creating plugins that create & configure network interfaces in Linux containers. Multus strictly adheres to the CNI specification described in the following link: https://github.com/containernetworking/cni/blob/master/SPEC.md

Operationally, Multus behaves as a broker and arbiter of other CNI plugins, meaning it invokes other CNI plugins (e.g. Flannel, Calico, SR-IOV, vHost CNI) to do the actual work of creating the network interfaces. When configuring Multus, one plugin must be identified as the master plugin and this is used to configure and manage the primary network interface (eth0).

Only information from the primary network interface is returned to Kubernetes after all networking is configured for a pod. Any number of additional CNI plugins can then be used to create additional network interfaces, but Kubernetes is not informed of the details related to those interfaces. This has the consequence that, while many network interfaces have been created and can be utilized within the pod, Kubernetes is not in a position to support services or security policy, for example, on those additional network interfaces.

To understand how Multus works, it is important to review how Kubernetes networking functions operate. Kubernetes uses network plugins to orchestrate networking. Currently there are two flavors of network plugin for Kubernetes:

1. **CNI plugins:** CNI plugins implement the CNI specification for interoperability of a container networking solution in Linux environment.
2. **Kubenet:** Kubenet implements a basic bridge cbr0 with host-local CNI plugin. It is also worth noting that kubenet is being deprecated, leaving CNI as the only supported framework.

The sequence diagram in Figure 2 shows the control flow for multiple network interface creation by the Multus CNI plugin. **Kubelet** is the primary agent that runs on each node in a Kubernetes cluster. Its main functions are to register the node with the Kubernetes control plane and to provide lifecycle management to any pods that are subsequently scheduled to run on that node. It is also responsible for establishing the network interfaces for each pod; Kubelet does this by reading the Multus CNI configuration file and then uses these configurations to set up each pod's network. In the setup in Figure 2, Kubelet is configured to use CNI as its networking plugin.

When Kubelet is invoked to set up a pod, it calls its container runtime (e.g. Docker or CoreOS Rocket (rkt)) to set up the pod. Kubelet also provides a network plugin wrapper to the container runtime to configure its network. In this case, it is the Multus CNI plugin. Multus can be used with a configuration file, using network objects or in combination of both. In any of these modes Multus reads its configuration and offloads the actual tasks of setting up the network to other CNI plugins called as delegates. Network objects are explained in section 4.0.

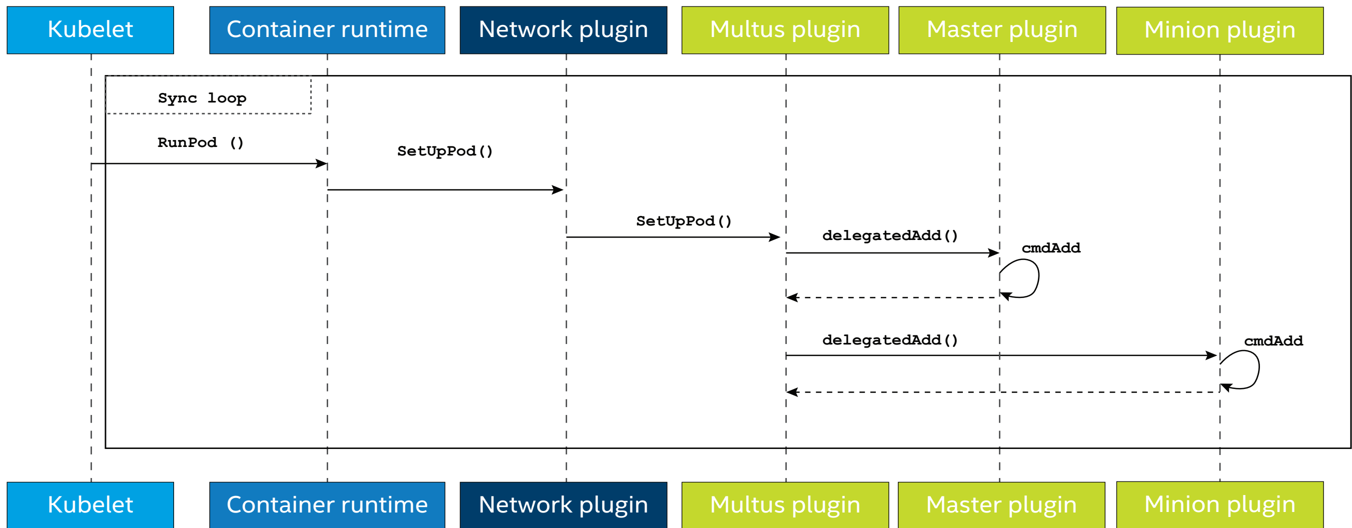


Figure 2. Multus workflow in Kubernetes

Multus then invokes **delegateAdd()** for each of these delegates (CNI plugins and their corresponding configurations). These, in turn, call their own **cmdAdd()** function to add a network interface for the Pod. These delegates define which CNI plugin to be invoked and what their parameters are. The arguments of these delegate CNI plugins can be stored as CRD or TPR object in Kubernetes.Refer to section 4.1 for more details.

Multus creates additional networks by taking network information that is supplied in the pod annotation. By evaluating the pod annotation, Multus will determine, which other CNI plugin should be invoked.

Note: The order of plugin invocation is important as it specifies the identity of the master plugin and that of the rest of the plugins, which are identified as minion plugins.

Multus enables support for NFV use cases that require multiple network interfaces and it also permits the use of interfaces and software stacks that would otherwise not be possible in Kubernetes, such as SR-IOV and the Data Plane Development Kit (DPDK).

The next section introduces accelerating the NFV Data plane with SR-IOV and DPDK.

3.0 Improving network performance using Multus and SR-IOV/DPDK

DPDK is an open source collection of libraries and drivers that support fast-packet processing by routing packets around the OS kernel and minimizing the number of CPU cycles needed to send and receive packets. DPDK libraries include multicore framework, huge page memory, ring buffers and poll mode drivers for networking and other network functions. For more information, go to www.dpdk.org.

SR-IOV is a PCI-SIG standardized method for isolating PCI Express (PCIe) native hardware resources for manageability and performance reasons. In effect, this means a single PCIe device, referred to as the physical function (PF), can appear as multiple separate PCIe devices, referred to as virtual functions (VF), with the required resource arbitration occurring in the device itself.

In the case of an SR-IOV-enabled network interface card (NIC), each VFs MAC and IP address can be independently configured and packet switching between the VFs occurs in the device hardware. The benefits of using SR-IOV networking devices in Kubernetes pods include:

- Direct communication with the NIC device allows for close to “bare-metal” performance.
- Support for multiple fast network packet processing simultaneous workloads in user space (based on DPDK, for example).
- Leveraging of NIC accelerators and offloads per workload.
- For more information, read the Intel whitepaper "SR-IOV for NFV Solutions." More details are in Appendix Table 7.5.

Intel introduced the SR-IOV CNI plugin to allow a Kubernetes pod to be attached directly to an SR-IOV virtual function (VF) in one of two modes. The first mode uses the standard SR-IOV VF driver in the container host’s kernel. The second mode supports DPDK VNFs that execute the VF driver and network protocol stack in user space.

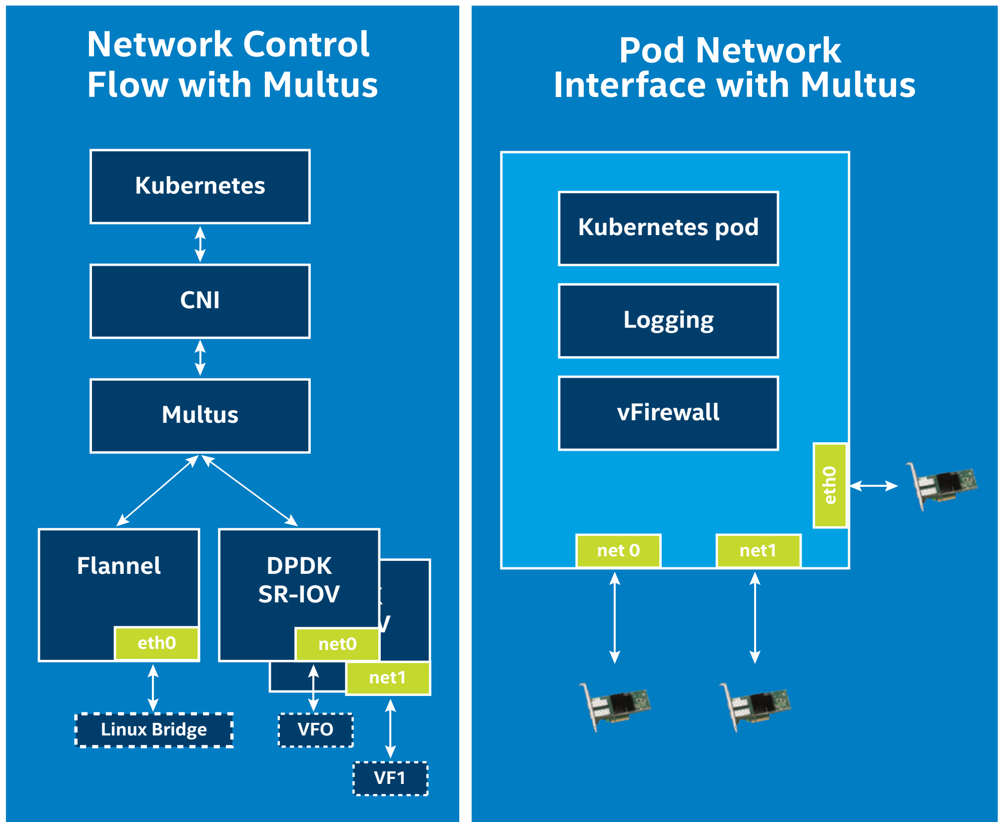


Figure 3. Multus networking with SR-IOV/ DPDK CNI

DPDK allows the application to achieve packet processing performance that greatly exceeds the ability of the kernel network stack. For performance benchmark results, please refer to "Kubernetes and Container Bare Metal on Intel Xeon Scalable Platform for NFV Use Cases." A link to this document can be found in the reference documents table 7.5 in the Appendix.

Let us consider the diagram in Figure 3 which shows a containerized virtual firewall with logging capability in a pod setup with three network interfaces. In this diagram, the eth0 is used as the management interface for the pod, which allows that pod to communicate with any other pod within the Kubernetes cluster. eth0 is the main networking interface in Kubernetes.

In addition, two more SR-IOV VF interfaces - net0 and net1 - are shown. These interfaces are created for the accelerating data plane networking. For example, a virtual firewall requires that two networks are isolated from each other using firewall rules. The VLAN technology is implemented between the virtual firewall and the 802.1Q switches and routers. The firewall recognizes VLAN IDs, and applies the firewall rules specific to each VLAN. This can include authenticating data or applying relevant policies established in the data plane network.

The advantages of this setup include:

- Logical segmentation of network
- Granular firewall rules specific to VLAN tagging
- Improved network throughput and low latency

The next two sections detail how Multus can be configured in Kubernetes with the SR-IOV/DPDK CNI plugin.

4.0 Configuring Multus in Kubernetes

Multus introduces the concept of network objects. A network object represents a network into which a pod interface is attached. They are logical references for a network that have a global scope. Network objects are considered as cluster-wide objects as they are stored in Kubernetes master registry.

This section describes the two Multus configuration options for selecting networks in Kubernetes:

- Configure Multus using network objects with default network
- Configure Multus using config file

The Multus configuration using network objects allows the user to select the network per pod instead of selecting the network per node. Below are some sample scenarios that the user might want to setup:

- Pod A spec with network object annotation “flannel” and “SRIOV” connected to flannel and SR-IOV networks.
- Pod B spec with network object annotation “Calico” connected to Calico network.
- Pod C spec without any network object annotation, but having “Weave” as default network, connected to Weave network.

Note: Multus configured to use config file works as any other CNI plugin.

The following section describes how to configure Multus with network object using a virtual firewall use case that involves invoking three networking interfaces as described in the section 3.0 (Figure 3). These include Flannel, SR-IOV and SR-IOV with virtual LAN (VLAN) tagging.

4.1 Configure Multus using network objects

A custom resource is an extension in Kubernetes that stores a collection of objects of a particular kind such as network objects. Custom resource definition (CRD) is a built-in feature in Kubernetes that offers a simple way to create custom resources. This mechanism provides a facility to describe a new API entity to the Kubernetes API server. CRD is the successor of Third Party Resource (TPR) from Kubernetes version 1.7 onwards. A CRD provides a stable object with the introduction to new features such as pluralization of resource names and the ability to create non-namespaced CRDs.

In this section, the "network” object is created as a custom resource using CRD. For more information using TPR to create objects, go to “How to define TPR-based network objects” section in in the Appendix under section 7.1. Multus uses these network objects to create network interfaces in Kubernetes.

Kubelet is responsible for establishing the network interfaces for each pod; it does this by invoking the configured CNI plugin. When Multus is invoked, it recovers pod annotations related to Multus. It then uses these annotations to recover a Kubernetes CRD object. The Kubernetes CRD is an object that informs the Kubelet of which plugins to invoke and the required configurations to be passed. The identity of the primary plugin as well as the order of plugin invocation is important. The flow chart of activities required to create Multus network interfaces in Kubernetes is demonstrated in Figure 4.

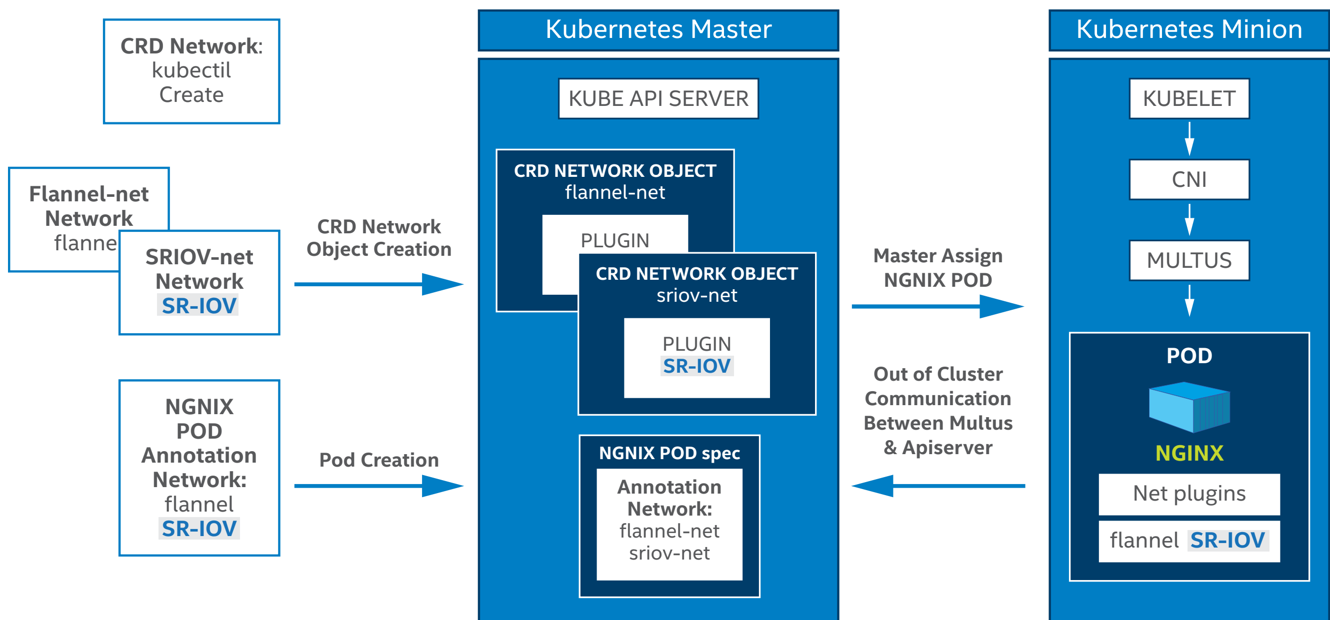


Figure 4. Flow chart of activities required to created Multus network interfaces in Kubernetes.

As mentioned previously, Multus is compatible with both CRD and TPR extension objects. However, TPR is only supported in Kubernetes versions up to v1.7. Later versions support only CRD-based objects. Network objects, therefore, need to be defined using CRD or TPR depending on which Kubernetes version in use. However, in both CRD/TPR-based network objects, client applications call the same API self-link (Kubelet, for example). This means the developer or the network orchestrator has a standard way to call the API regardless of whether a CRD or TPR network object called. Refer to section 7.1 in the Appendix for more information on TPR's.

To setup multiple network interfaces, the user needs to first define the required network objects and then create them. Instructions for defining and creating the CRD network objects are described below:

4.1.1 Defining CRD-based network objects

1. First, create a CRD network object specification as shown in the following steps and save it as a "crdnetwork.yaml" file:

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <plural>.<group>
  name: networks.kubernetes.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: kubernetes.com
  # version name to use for REST API: /apis/<group>/<version>
  version: v1
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    # plural name to be used in the URL: /apis/<group>/<version>/<plural>
    plural: networks
    # singular name to be used as an alias on the CLI and for display
    singular: network
    # kind is normally the CamelCased singular type. Your resource manifests use this.
    kind: Network
    # shortNames allow shorter string to match your resource on the CLI
    shortNames:
    - net
```

2. Then run the following kubectl command to create the Custom Resource Definition:

```
# kubectl create -f ./crdnetwork.yaml
customresourcedefinition "network.kubernetes.com" created
```

3. Run the following kubectl command to check that Network CRD is created:

```
# kubectl get CustomResourceDefinition
NAME                     KIND
networks.kubernetes.com CustomResourceDefinition.v1beta1.apiextensions.k8s.io
```

4. After that, save the following YAML to flannel-network.yaml:

```
apiVersion: "kubernetes.com/v1"
kind: Network
metadata:
  name: flannel-networkobj
plugin: flannel
args: '[
        {
                "delegate": {
                        "isDefaultGateway": true
                }
        }
]'
```

5. Now, create the custom resource definition:

```
# kubectl create -f customCRD/flannel-network.yaml
network "flannel-networkobj" created
# kubectl get network
NAME                 KIND                               ARGS          PLUGIN
flannel-networkobj   Network.v1.kubernetes.com    [ { "delegate": { "isDefaultGateway": true }
} ]   flannel
```

6. Then, get the custom network object details:

```
# kubectl get network flannel-networkobj -o yaml
apiVersion: kubernetes.com/v1
args: '[ { "delegate": { "isDefaultGateway": true } } ]'
kind: Network
metadata:
  clusterName: ""
  creationTimestamp: 2017-07-11T21:46:52Z
  deletionGracePeriodSeconds: null
  deletionTimestamp: null
  name: flannel-networkobj
  namespace: default
  resourceVersion: "6848829"
  selfLink: /apis/kubernetes.com/v1/namespaces/default/networks/flannel-networkobj
  uid: 7311c965-6682-11e7-b0b9-408d5c537d27
plugin: flannel
```

4.1.3 How to create network objects

After completing the steps in the previous section, the CRD network object definition will have been added to the API server. It is now possible to create the network objects. Network objects should contain the network args parameter in JSON format. In the following example, the plugin and args fields are set to the object of kind *Network*. The object of kind *Network* is derived from the metadata.name of the CRD object that was in created previous steps.

1. First, save the following YAML to file flannel-network.yaml:

```
apiVersion: "kubernetes.com/v1"
kind: Network
metadata:
  name: flannel-conf
plugin: flannel
args: '[
        {
                "delegate": {
                        "isDefaultGateway": true
                }
        }
]'
```

2. Now, run kubectl create command to create flannel-conf network object:

```
# kubectl create -f ./flannel-network.yaml
network "flannel-conf" created
```

3. Then, verify the network objects using kubectl:

```
# kubectl get network
NAME                KIND
flannel-conf        Network.v1.kubernetes.com
```

4. It is also possible to view the raw JSON data of the network objects. The instructions that follow show the custom plugin and args fields from the yaml file that the network object was created with:

```
# kubectl get network flannel-conf -o yaml
apiVersion: kubernetes.com/v1
args: '[ { "delegate": { "isDefaultGateway": true } } ]'
kind: Network
metadata:
  creationTimestamp: 2017-06-28T14:20:52Z
  name: flannel-conf
  namespace: default
  resourceVersion: "5422876"
  selfLink: /apis/kubernetes.com/v1/namespaces/default/networks/flannel-conf
  uid: fdcb94a2-5c0c-11e7-bbeb-408d5c537d27
plugin: flannel
```

5. The plugin field should be the name of the CNI plugin and args should have the Flannel args, it should be in the JSON format as shown above. Network objects for Calico, Weave, Romana, & Cilium can also be created similarly.

6. Save the following YAML to sriov-network.yaml:

```
apiVersion: "kubernetes.com/v1"
kind: Network
metadata:
  name: sriov-conf
plugin: sriov
args: '[
       {
              "if0": "enp12s0f1",
              "ipam": {
                 "type": "host-local",
                 "subnet": "10.56.217.0/24",
                 "rangeStart": "10.56.217.171",
                 "rangeEnd": "10.56.217.181",
                 "routes": [
                     { "dst": "0.0.0.0/0" }
                 ],
                 "gateway": "10.56.217.1"
             }
       }
]'
```

7. After that, save the following YAML to file sriov-vlanid-l2enable-network.yaml:

```
apiVersion: "kubernetes.com/v1"
kind: Network
metadata:
  name: sriov-vlanid-l2enable-conf
plugin: sriov
args: '[
       {
                "if0": "enp2s0",
                "vlan": 210,
                "if0name": "north",
                "l2enable": true
        }
]'
```

8. Complete the following two steps to create the network object “sriov-vlanid-l2enable-conf” and “sriov-conf”:

```
# kubectl create -f ./sriov-vlanid-l2enable-network.yaml
network "sriov-vlanid-l2enable-conf" created

# kubectl create -f ./sriov-network.yaml
network "sriov-conf" created
```

9. Finally, verify the network objects using kubectl:

```
# kubectl get network
NAME                         KIND
flannel-conf        Network.v1.kubernetes.com
sriov-vlanid-l2enable-conf
Network.v1.kubernetes.com
sriov-conf          Network.v1.kubernetes.com

# systemctl restart kubelet
```

At this stage, the network resources have been configured and created. The next step is to deploy pods using these network resources.

4.1.4 Deploy pods with multiple interfaces

1. Create a sample pod specification **pod-multi-network.yaml** file with following contents. In this case flannel-conf network object act as the primary network:

```
# cat pod-multi-network.yaml

apiVersion: v1
kind: Pod
metadata:
  name: multus-multi-net-pod
  annotations:
    networks: '[
        { "name": "flannel-conf" },
        { "name": "sriov-conf"},
     { "name": "sriov-vlanid-l2enable-conf" }
    ]'
spec:  # specification of the pod's contents
  containers:
  - name: multus-multi-net-pod
    image: "busybox"
    command: ["top"]
    stdin: true
    tty: true
```

2. Next, create multiple network-based pods from the master node:

```
# kubectl create -f ./pod-multi-network.yaml
pod "multus-multi-net-pod" created
```

3. Lastly, retrieve the details of the running pod from the master:

```
# kubectl get pods
NAME                    READY         STATUS        RESTARTS      AGE
multus-multi-net-pod    1/1           Running       0             30s
```

4.2 Configure Multus using config file

Multus can be configured to read the network configuration from its config file instead of network objects. In this case, all pods in the node will have the same network interface.

4.2.1 Multus configuration parameters

Multus accepts the configuration parameters in JSON format described in Table 1. Some of these parameters are required and thus need to be provided by user when configuring a Kubernetes minion node.

Table 1. Multus config parameters

Parameter Name	Type	Required	Description
name	string	Yes	The name of the network
type	string	Yes	"multus"
kubeconfig	string	No	kubeconfig file for the out of cluster communication with kube-apiserver
delegates	map	Yes	Delegate objects (underlying CNI plugin configuration). This is ignored if kubeconfig is added
masterplugin	bool	Yes	master plugin to report back the IP address and DNS to the container

Instructions on how to configure Multus are as follows:

4.2.2 Configure using CNI config file

1. Create Multus CNI configuration file /etc/cni/net.d/multus-cni.conf with the contents below in minions. Use only the absolute path to point to the kubeconfig file (it may change depending upon Kubernetes cluster env). This assumes all CNI binary files are located in in \opt\cni\bin directory (default location):

```
{
    "name": "minion-cni-network",
    "type": "multus",
    "kubeconfig": "/etc/kubernetes/node-                    kubeconfig.yaml"
}
```

2. Now, restart kubelet service:

```
# systemctl restart kubelet
```

4.2.3 Configure using the kubeconfig with default network

3. Certain automatic Kubernetes deployment models or wrapper programs require default networking features, in the event that the network object is absent from the pod specification. One example is an automated Ansible script and wrapper program that have been developed by a third party for non-Multus deployments. All Ansible scripts should work with Multus by utilizing Weave or another default networking function. Another example is in the following configuration where Weave acts as the default network in the absence of network field in the pod metadata annotation.

```
{
    "name": "minion-cni-network",
    "type": "multus",
    "kubeconfig": "/etc/kubernetes/node-kubeconfig.yaml",
    "delegates": [{
        "type": "weave-net",
        "hairpinMode": true,
        "masterplugin": true
    }]
}
```

4. Now, restart kubelet service:

```
# systemctl restart kubelet
```

4.3 Verifying pod networks

Once the configuration is complete, you can verify the pod networks are working as expected. The pod created with the specification defined in section 4.1.3 should have created three interfaces with the provided configurations. To verify these configurations, complete the following steps:

1. Run “ifconfig” command inside the container:

```
# kubectl exec -it multus-multi-net-poc – ifconfig

eth0  Link encap:Ethernet  HWaddr                  06:21:91:2D:74:B9
  inet addr:192.168.42.3  Bcast:0.0.0.0    Mask:255.255.255.0
  inet6 addr: fe80::421:91ff:fe2d:74b9/64            Scope:Link
  UP BROADCAST RUNNING MULTICAST  MTU:1450 Metric:1
  RX packets:0 errors:0 dropped:0                    overruns:0 frame:0
  TX packets:8 errors:0 dropped:0                    overruns:0 carrier:0
  collisions:0 txqueuelen:0
  RX bytes:0  (0.0 B)  TX bytes:648  (648.0 B)

lo    Link encap:Local Loopback
  inet addr:127.0.0.1  Mask:255.0.0.0
  inet6 addr: ::1/128 Scope:Host
  UP LOOPBACK RUNNING  MTU:65536  Metric:1
  RX packets:0 errors:0 dropped:0 overruns:0            frame:0
  TX packets:0 errors:0 dropped:0 overruns:0            carrier:0
  collisions:0 txqueuelen:1
  RX bytes:0  (0.0 B)  TX bytes:0  (0.0 B)

net0  Link encap:Ethernet                                  HWaddrD2:94:98:82:00:00
  inet addr:10.56.217.171  Bcast:0.0.0.0                Mask:255.255.255.0
  inet6 addr: fe80::d094:98ff:fe82:0/64                 Scope:Link
  UP BROADCAST RUNNING MULTICAST  MTU:1500              Metric:1
  RX packets:2 errors:0 dropped:0 overruns:0            frame:0
```

```
  TX packets:8 errors:0 dropped:0 overruns:0             carrier:0
  collisions:0 txqueuelen:1000
  RX bytes:120(120.0 B) TX bytes:648(648.0 B)

north  Link encap:Ethernet  HWaddr BE:F2:48:42:83:12
  inet6 addr: fe80::bcf2:48ff:fe42:8312/64           Scope:Link
  UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
  RX packets:1420 errors:0 dropped:0 overruns:0 frame:0
  TX packets:1276 errors:0 dropped:0 overruns:0 carrier:0 collisions:0 txqueuelen:1000
  RX bytes:95956 (93.7 KiB)  TX bytes:82200 (80.2 KiB)
```

As shown in the output screen above, there are three interfaces created along with a loopback interface.

Interface Name	Description
lo	Loopback
eth0@if41	Flannel network tap interface
net0	VF0 of NIC 1 assigned to the container by SR-IOV CNI plugin
north	VF0 of NIC 2 assigned with VLAN ID 210 to the container by SR-IOV CNI plugin

The description of the pod interfaces can be found in the following table:

2. It can be verified that the VLAN ID of the VF that is assigned from the SR-IOV NIC matches with the VLAN tag given in file **sriov-vlanid-l2enable-network.yaml** file in section 4.1.2 step 8. This is confirmed by checking the SR-IOV NIC information using IPROUTE2 utility as shown below:

```
# ip link show enp2s0

20: enp2s0: <BROADCAST,MULTICAST,UP,LOWER _ UP> mtu 1500 qdisc mq state UP mode DEFAULT group
  default qlen 1000
  link/ether 24:8a:07:e8:7d:40 brd ff:ff:ff:ff:ff:ff
  vf 0 MAC 00:00:00:00:00:00, vlan 210, spoof checking off, link-state auto
  vf 1 MAC 00:00:00:00:00:00, vlan 4095, spoof checking off, link-state auto
  vf 2 MAC 00:00:00:00:00:00, vlan 4095, spoof checking off, link-state auto
  vf 3 MAC 00:00:00:00:00:00, vlan 4095, spoof checking off, link-state auto
```

After completing the steps above, a Kubernetes pod should be configured with three network interfaces: “eth0”, “net0” and “north” along with a loopback interface “lo”. In the above instruction, the multi networking in Kubernetes is showcased with network object creation via the Multus CNI plugin. Please refer to the GitHub link for more information: https://github.com/Intel-Corp/multus-cni/

5.0 Configuring SR-IOV/DPDK in Kubernetes

SR-IOV-enabled NICs allow sharing a physical NIC port transparently amongst many VNFs in many virtual environments. Each VF can be assigned to one container, and configured with separate MAC, VLAN and IP. The SR-IOV CNI plugin enables the Kubernetes pods to attach to an SR-IOV VF. The plugin looks for the first available VF on the designated port in the Multus configuration file. The plugin also supports the DPDK driver (i.e. vfio-pci) for these VFs. The DPDK driver can provide high-performance networking interfaces to the Kubernetes pods for data plane acceleration for the containerized VNFs.

5.1 Enable SR-IOV

The following steps enable the SR-IOV plugin for Intel ixgbe NIC on CentOS, Fedora or RHEL..

1. First, enable SR-IOV using the following command:

```
# vi /etc/modprobe.conf
options ixgbe max _ vfs=8,8
```

2. Then, provide additional network configuration parameters for the SR-IOV plugin as documented in in section 4.1 step 7. The “**args**” field in sriov-network.yaml file holds the SR-IOV configuration parameter values. These parameters are explained below:
    - **name** (string, required): the name of the network
    - **type** (string, required): "sriov"
    - **if0** (string, required): name of the PF
    - **l2enable** (string, optional): SR-IOV interface without IP address
    - **ipam** (dictionary, required for Kernel mode): IPAM configuration to be used for this network (kernel mode). Refer to IPAM for more information.

- **dpdk** (dictionary required only in userspace)
  - **kernel_driver** (string, required for DPDK mode): name of the NIC driver e.g i40evf
  - **dpdk_driver** (string, required for DPDK mode): name of the DPDK driver e.g. vfio-pci
  - **dpdk_tool** (string, required for DPDK mode): absolute path of dpdk bind script e.g. dpdk-devbind.py Extra arguments
  - **vf** (int, optional): VF index, default value is 0
  - **vlan** (int, optional): VLAN ID for VF device Usage in kernel mode using IPAM

3. After setting these parameters, create the network objects using the SR-IOV plugin as outlined in step 8 in section 4.1.

The SR-IOV Plugin can be used with DPDK or with the kernel. By default, the plugin runs in kernel mode. In order to run in DPDK mode, the following network parameter needs to be set:

- dpdk (dictionary required only in userspace)
  - kernel_driver (string, required for DPDK mode): name of the NIC driver e.g i40evf
  - dpdk_driver (string, required for DPDK mode): name of the DPDK driver e.g. vfio-pci

An explanation of both modes is given in section 5.1.1 and 5.1.2 below.

5.1.1 Kernel mode:

The SR-IOV CNI plugin gets the SR-IOV VF interface from the host network namespace to the container network namespace and assigns the IPAM information to the SR-IOV VF interface.

5.1.2 DPDK mode:

The SR-IOV CNI plugin gets the SR-IOV VF interface and binds the interface to the DPDK user space. During this process, the PCI address is stored in the host, and the plugin makes an ongoing effort to be stateless. During the deletion process, the SR-IOV CNI plugin retrieves the PCI address and unbinds the SR-IOV VF interface from the DPDK user space to kernel space.

5.1.3 Configuring Multus with Flannel and DPDK –SR-IOV CNI plugins

To create a Multus CNI configuration file /etc/cni/net.d/multus-cni.conf with the content below in a Kubernetes node, make sure that the DPDK driver is loaded and the SR-IOV VF is created. Be sure to use an absolute path when setting the dpdk_tool option.

1. Follow these steps to begin configuration:

```
{
   "name": "minion1-multus-demo-network",
   "type": "multus",
   "delegates": [
       {
           "type": "sriov",
           "if0": "enp4s0f3",
           "if0name": "north0",
           "dpdk": {
               "kernel _ driver":"ixgbevf",
               "dpdk _ driver":"igb _ uio",
               "dpdk _ tool":"/root/dpdk/                                  tools/dpdk-devbind.py"
           }
       },
       {
           "type": "flannel",
           "masterplugin": true,
           "delegate": {
               "isDefaultGateway": true
           }
       }
    ]
}
```

Above is an example Multus CNI plugin configuration file, both the flannel and DPDK-SR-IOV CNI plugins are called to provide the actual CNI networking.

6.0 Conclusion

Having multiple network interfaces in a Kubernetes pod is an essential feature for many VNF applications. This can be accomplished today through the use of the Multus CNI and the other open source software components described in this document. Additionally, the availability of multiple network interfaces makes improved network throughput for container-based applications possible through the use of interfaces to SR-IOV and DPDK. Intel has contributed to the development of these technologies as part of its support for virtualized computing.

For more information on what Intel is doing with containers, go to https://networkbuilders.intel.com/network-technologies/intel-container-experience-kits.

7.0 Appendix

This appendix includes:

- information on how to define TPR-based network objects
- tables that show the hardware platform and software referenced in this document.
- a helpful summary of the acronyms used in this document
- links to reference documents

7.1 How to define TPR-based network objects

1. Start by creating a third-party resource “tprnetwork.yaml” for the network object as shown below:

```
apiVersion: extensions/v1beta1
kind: ThirdPartyResource
metadata:
  name: network.kubernetes.com
description: "A specification of a Network obj in the kubernetes"
versions:
- name: v1
```

2. Then, run a kubectl create command for the third-party resource:

```
# kubectl create -f ./tprnetwork.yaml
thirdpartyresource "network.kubernetes.com" created
```

3. Next, run kubectl get command to check the Network TPR creation:

```
# kubectl get thirdpartyresource
NAME                     DESCRIPTION
network.kubernetes.com   A specification of a Network obj in the kubernetes
VERSION(S)
v1
```

7.2 Hardware

Table 7.2 Hardware ingredients used in performance tests

Item	Description	Notes
Platform	Intel® Server Board S2600WFQ	Intel® Xeon® processor-based dual-processor server board with 2 x 10 GbE integrated LAN ports
Processor	2 Intel® Xeon® Gold Processor 6138T	2.0 GHz; 125 W; 27.5 MB cache per processor 20 cores, 40 hyper-threaded cores per processor
Memory	192GB Total; Micron MTA36ASF2G72PZ	12x16GB DDR4 2133MHz 16GB per channel, 6 Channels per socket
NIC	Intel® Ethernet Network Adapter XXV710-DA2 (2x25G)	2 x 1/10/25 GbE ports, Firmware version 5.50
Storage	Intel DC P3700 SSDPE2MD800G4	SSDPE2MD800G4 800 GB SSD 2.5in NVMe/PCIe
BIOS	Intel Corporation SE5C620.86B0X.01.0007.060920171037 Release Date: 06/09/2017	Hyper-Threading – Enable Boot performance Mode – Max Performance Energy Efficient Turbo – Disabled Turbo Mode – Disabled C State – Disabled P State – Disabled Intel VT – x Enabled Intel VT – d Enabled

7.3 Software

Table 7.3 Software ingredients used in performance tests

Software Component	Description	References
Host Operating System	Ubuntu 16.04.2 x86_64 (Server) Kernel: 4.4.0-62-generic	https://www.ubuntu.com/download/server
NIC Kernel Drivers	i40e v2.0.30 i40evf v2.0.30	https://sourceforge.net/projects/e1000/files/i40e%20stable
DPDK	DPDK 17.05 (Software download)	http://fast.dpdk.org/rel/dpdk-17.05.tar.xz
CMK	v1.1.0 & v1.2.1	https://github.com/Intel-Corp/CPU-Manager-for-Kubernetes
Ansible	Ansible 2.3.1.0	https://github.com/ansible/ansible/releases
Bare Metal Container Setup scripts	Includes Ansible* scripts to deploy Kubernetes v1.6.6 & 1.8.4	https://github.com/intel/container-experience-kits
Docker	v1.13.1	https://docs.docker.com/engine/installation/
SR-IOV-CNI	v0.2-alpha. commit ID: a2b6a7e03d8da456f3848a96c6832e6aefc968a6	https://www.ubuntu.com/download/server

7.4 Terminology

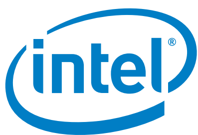
Table 7.4 Terminology

Term	Description
CNCF	Cloud Native Computing Foundation
CNI	Container Network Interface
CNVNF	Cloud Native Virtualized Network Functions
COE	Container Orchestration Engine
CommSP	Communications Service Provider
CRD	Custom Resource Definition
DPDK	Data Plane Development Kit
IPAM	IP address Management
IPSec	Encryption Protocol for IP Networks
LF	Linux Foundation
NFD	Node Feature Discovery
NFV	Network Functions Virtualization
NIC	Network interface card
PCIe	Peripheral Component Interconnect Express
PF	Physical Function
Pod	A Group of One Or More Containers in Kubernetes
rkt	CoreOS Rocket
SDN	Software Defined Network
SLA	Service Level Agreement
SR-IOV	Single-Root Input/Output Virtualization
STDIN	Standard input
TPR	Third Party Resource
VETH	Virtual Ethernet interface
VF	Virtual Function
VLAN	Virtual LAN
VM	Virtual Machine
VNF	Virtual Network Function
VPP	Vector Packet Processing

7.5 Reference documents

Table 7.5 Reference documents

Document	Document No./Location
Deploying Kubernetes and Container Bare Metal Platform for NFV Use Cases with Intel® Xeon® Scalable Processors	https://networkbuilders.intel.com/network-technologies/container-experience-kits
Kubernetes and Container Bare Metal on Intel Xeon Scalable Platform for NFV Use Cases	https://networkbuilders.intel.com/network-technologies/container-experience-kits
Enabling New Features with Kubernetes for NFV	https://networkbuilders.intel.com/network-technologies/container-experience-kits
NFV Reference Design for A Containerized vEPC application	https://builders.intel.com/docs/networkbuilders/nfv-reference-design-for-a-containerized-vepc-application.pdf
Understanding CNI (Container Networking Interface)	http://www.dasblinkenlichten.com/understanding-cni-container-networking-interface/
SR-IOV for NFV Solutions (PDF download)	https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/sr-iov-nfv-tech-brief.pdf



By using this document, in addition to any agreements you have with Intel, you accept the terms set forth below.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifi cations. Current characterized errata are available on request. Contact your local Intel sales once or your distributor to obtain the latest specifications and before placing your product order.

Intel technologies may require enabled hardware, specific software, or services activation. Check with your system manufacturer or retailer. Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/performance.

All products, computer systems, dates and gestures specified are preliminary based on current expectations, and are subject to change without notice. Results have been estimated or simulated using internal Intel analysis or architecture simulation or modeling, and provided to you for informational purposes. Any differences in your system hardware, software or configuration may affect your actual performance.

No computer system can be absolutely secure. Intel does not assume any liability for lost or stolen data or systems or any damages resulting from such losses.

Intel does not control or audit third-party websites referenced in this document. You should visit the referenced website and confirm whether referenced data are accurate.

Intel Corporation may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights that relate to the presented subject matter. The furnishing of documents and other materials and information does not provide any license, express or implied, by estoppel or otherwise, to any such patents, trademarks, copyrights, or other intellectual property rights.

Intel, the Intel logo, Intel vPro, and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.

© 2018 Intel® Corporation Printed in USA Multiple Network Interfaces in Kubernetes Application Note 04/18/HM/DJA/PDF002 Please Recycle SKU 336866-002US